Ejercicios de Programación

Variables y Operadores

1. Crea un algoritmo que pida al usuario su nombre, su DNI y fecha de nacimiento y a continuación muestre esa información por pantalla de manera ordenada. Pej: Si el usuario se llama *Juan*, su DNI es *56789123D* y nació el *23/02/1998* se mostrará:

```
Nombre.............: Juan
DNI................: 56789123D
Fecha de Nacimiento..: 23/02/1998
```

2. Solicita dos números por teclado. Muestra por pantalla la suma, la resta, la multiplicación, la potencia y la división de ambos números. Usa la siguiente plantilla:

```
La suma de __ y __ es:
La resta de __ y __ es:
La multiplicación de __ y __ es:
El cociente de __ entre __ da:
__ elevado a __ da:
```

3. Crea un algoritmo que calcule la media de 5 números que se le pasan por teclado.

4. Diseña un algoritmo que pida dos números por teclado (m y n) y calcule las siguientes expresiones:

a) `m / n*(m-n)`

b) `12.3 / m + 5 - n * 9`

c) $m * 2048 / n*1024 - m^n$

d) El resto de la división: $(3203 /m-n)^n / (n^2*m^3)$

5. Diseña un algoritmo que pida por teclado un tiempo expresado en segundos y muestre por pantalla ese valor expresado en horas, minutos y segundos.

6. Crea un algoritmo que, partiendo de la cadena `"LA LLUVIA EN SEVILLA ES UNA MARAVILLA"` obtenga las cadenas `"sevilla es una maravilla"` y `"la lluvia es una maravilla"`. Muestra ambas por pantalla.

7. Para la cadena `cad = "Volando, volando… siempre arriba"`, crear un algoritmo que nos indique la posición de la letra 'd' en los 7 primeros caracteres de `cad`.

b) Igual pero ahora busca la letra 'd' en los 7 últimos

8. Crea un algoritmo que simule el lanzamiento de un dado de N caras. El número de caras del dado se solicitará por teclado.

9. Crea un algoritmo que sirva para resolver ecuaciones de primer grado del tipo: `ax+b=0` Vamos a suponer que `a` y `b` no van a ser cero nunca.

10. Crea un algoritmo que sirva para resolver ecuaciones de segundo grado del tipo: $ax^2+bx+c=0$ Vamos a suponer que `a,b` y `c` no van a ser cero nunca.

11. Crea un programa que, tomando como dato por teclado la cantidad de Mbps que tiene una linea, calcule cual será la descarga máxima de esa línea en MiB por segundo.

12. Codifica un programa que calcule la conversión de:

a) MegaBytes (MB) a MebiBytes (MiB)

b) MebiBytes (MiB) a Megabits (Mb)

13. Se necesita un algoritmo que calcule el coste total de baldosas cuadradas necesarias para cubrir el suelo de una habitación rectangular. El programa solicitará como datos las dimensiones de la habitación en metros (ancho y largo), el lado de cada baldosa expresado en centímetros y el precio por baldosa expresado en euros.

Con esos datos calculará el número de baldosas necesarias que hay que colocar en la habitación (redondeando hacia arriba si el resultado no es un numero entero) y el precio total a invertir.

14. Un partido de fútbol tiene una duración de 90 minutos. El minuto 1 se considera que abarca desde los 0 segundos hasta los 59 segundos. El minuto 2 abarca desde los 60 segundos hasta los 119 segundos... así sucesivamente hasta el último minuto, que es el minuto 90 y abarca desde los 5340 segundos hasta los 5400 segundos.

Crea un programa que pida al usuario el número de segundos transcurridos y muestre el minuto en que nos encontramos.

15. Pedir por teclado un número entero de 4 cifras.

a) Mostrar cada una de sus cifras (una debajo de otra)

b) Crear un nuevo número con las cifras del primero pero al revés.

Nota: debe leerse por teclado un número entero NO una cadena.

Secuencias Condicionales

16. Se necesita un algoritmo que calcule la media de de tres notas introducidas por teclado y en caso de que esa media sea superior a 6.5, se muestre por pantalla *'Promocionado con una media de XX'*. En caso contrario, mostrar *'No promocionas'*.

17. Solicitar que el usuario introduzca una clave dos veces. Mostrar un mensaje indicando si las claves son iguales o si son diferentes.

18. Realizar un algoritmo que pida por teclado dos números. Si el primero es mayor al segundo, mostrar la resta y la división de ambos. Si el segundo es mayor o igual al primero, mostrar la suma y el producto de ambos.

19. Diseña un algoritmo que calcule el mayor de 5 números introducidos por el usuario. Utiliza la versión de el "candidato" para resolverlo. Hay que controlar además, que los valores de introducidos sean correctos: números positivos.

20. Se pide realizar un algoritmo que solicite por teclado tres números. Si todos los valores ingresados son menores a 0, se mostrará por pantalla:

`Los números son:` *`números introducidos`*

`El mayor es:` *`el número mayor`*

`El menor es:` *`el número menor`*

21. Realiza un algoritmo que indique si un año es bisiesto o no. Un año es bisiesto si es divisible entre 4 y no es divisible entre 100 o es divisible entre 400.

22. Necesitamos realizar un algoritmo que dados un mes (número del 1 al 12) y un año introducidos por el usuario, indique si ese mes tiene 31, 30, 29 o 28 días. Recuerda que en los años bisiestos, Febrero tiene 29 días y no 28.

23. Completa el ejercicio 10 de esta relación (ecuaciones de segundo grado) contemplando la posibilidad de que `a`,`b` o `c` puedan ser cero.

24. Pregunta el nombre de dos jugadores. A continuación, simula una partida de tirada de dados: el primer jugador tira un dado de 6 caras y saca una puntuación. Luego hace lo mismo el segundo jugador. Al final se indica que jugador ha ganado.

25. Realiza un programa que indique si un numero introducido por teclado es de 5 cifras y además es par.

26. El sistema de verificación en dos pasos funciona de la siguiente manera:

- Se le pide al usuario que introduzca su nick, su email y la contraseña.
- Si el email y la contraseña son correctos (los datos se dicen en clase), el programa va a generar un número aleatorio de 4 cifras y lo va a mostrar por pantalla.
- A continuación se vuelve a pedir el email, la contraseña y, esta vez, el número aleatorio.
- Si todo ha ido correctamente, se mostrará el mensaje: "*Verificación aceptada. Bienvenid@: xxxxxx* (siendo xxxxxx el nick del usuario).
- Si hay algún fallo, se mostrará un mensaje de error.

Crea un algoritmo que reproduzca el comportamiento antes indicado.

27. Una empresa maneja números enteros positivos de 4 dígitos para controlar sus productos. Estos números se denominan POOC:

- P representa una de 8 provincias.
- OO un tipo de operación.
- C es un dígito de control.

Escribe un programa que lea por teclado un número entero y escriba en pantalla los dígitos desglosados si es un número POOC válido, o un mensaje de error si no es un número POOC válido. Para ser válido se debe cumplir las 3 condiciones siguientes:

- El número debe ser exactamente de 4 cifras.
- El número de provincia debe ser un número entre 1 y 8 (ambos incluidos).
- El dígito de control tiene que ser igual al resto de la división entera del tipo de operación entre el número de provincia.

28. Realiza un programa que dada una cantidad de dinero en Euros, realice un desglose en billetes y monedas. Ej:

```
Introduce cantidad a desglosar: 434
El desglose obtenido es...
2 billetes de 200€
1 billete de 20€
1 billete de 10€
2 monedas de 2€
```

Los billetes disponibles son de 500,200,10,50,20,10 y 5€ y las monedas de 2 y 1€.

Bucles

29. Mejora el ejercicio 14 (minuto partido de futbol) añadiendo lo siguiente:

- Impiden la entrada de segundos máximos (5400). Si el usuario se pasa, saca un mensaje por pantalla indicándolo y vuelve a pedir el número.
- El programa no termina hasta que el usuario mete un numero negativo.

30. Escribe un programa que lea de teclado 2 números enteros y saque en pantalla todos los números que estén entre ellos. Ejemplo:

```
Introduce primer número: 4
Introduce segundo número: 10
4,5,6,7,8,9,10
```

Importante: El programa no debe asumir que el usuario introducirá primero el número más pequeño.

b) Modifica el programa para que solo escriba en pantalla los números pares del intervalo.

31. Realizar un algoritmo que pida por teclado un numero entero y a continuación muestre todos los divisores de ese número.

32. Escriba un programa que pida un número al usuario. Si el número introducido no es divisible por 2 y 3 entonces el programa mostrará un mensaje de error y volverá a pedir un número al usuario. En caso de que el número sea divisible por dichos números se mostrará el resultado de dividirlo por ellos y se terminará el programa.

33. Realizar un programa que calcule el resultado de elevar un número a otro. Para ello, leerá dos números enteros, la base y el exponente, y calculará el resultado, mostrándolo en pantalla. No se pueden utilizar las funciones que te calculen automáticamente el resultado como, por ejemplo, la función Math.pow.

34. Crea un programa que muestre todos los múltiplos de 6 entre 6 y 200, ambos inclusive.

35. Implementa un programa que solicite al usuario dos números: *n* y *m*. A continuación debe mostrar todos los múltiplos de *n* entre *n* y *m*n.* Ejemplo:

```
Introduce primer número: 4
Introduce segundo número: 10
Los múltiplos de 4 entre 4 y 40 son: 8,12,16,20,24,28,32,36,40
```

36. Desarrolla un algoritmo que pida 5 elementos por teclado y continuación muestre la suma de todos los elementos, la suma de aquellos números mayores a 36 y la suma de aquellos números menores a 10. *Nota: no puedes usar 5 variables.*

37. Escribe un algoritmo que pida una lista de números enteros uno a uno hasta que se introduzca el valor 0. A continuación debe escribir por pantalla la posición de la primera y de la última aparición del número 12 dentro de la lista. Ejemplo:

```
Dame un número: 8
Dame un número: 9
Dame un número: 12
Dame un número: 7
Dame un número: 6
Dame un número: 5
Dame un número: 4
Dame un número: 12
Dame un número: 3
Dame un número: 2
Dame un número: 1
Dame un número: 0

Primera aparición en posición 3

Última aparición en posición 8
```

Importante: Si el número 12 no está en la lista, se mostrará el valor 0 y si el número 12 aparece sólo una vez, tanto la primera como la última posición serán el mismo valor.

b) Mostrar también por pantalla la lista de números introducida.

38. Queremos realizar un algoritmo que vaya pidiendo una sucesión de notas por teclado hasta que se introduzca el valor -1. Tras eso se mostrará la media ponderada de las notas introducidas. Importante: Hay que controlar que los valores de las notas introducidas sea el correcto (entre 0 y 10)

b) Realiza una variante del ejercicio anterior donde el primer valor que se solicita indica el número total de calificaciones que se deben introducir.

39. Diseña un algoritmo que calcule el factorial de un número pedido por teclado. Pej: el factorial de 5 es: `5! = 5*4*3*2*1 = 120`

40. Diseña un algoritmo que pida dos números por teclado (m y n) y calcule el número combinatorio de ambos.
El número combinatorio se calcula usando la expresión: `m! / n!*(m-n)!`
Importante: Hay que obligar a que `m >= n`.

41. Algoritmo que solicite al usuario un numero entero y a continuación muestre la tabla de multiplicar de ese número hasta el 10. Pej:

```
Tabla de multiplicar de? 3
3x1=3
3x2=6
3x3=9
3x4=12
3x5=15
3x6=18
3x7=21
3x8=24
3x9=27
3x10=30
```

42. Escribe un algoritmo que lea por teclado un número N entero positivo y dibuje un cuadrado de asteriscos de lado N. Pej: Si el número es 4, se obtiene:

```
* * * *
* * * *
* * * *
* * * *
```

43. Escribe un algoritmo que lea por teclado un número N entero positivo y dibuje un triangulo de asteriscos con base y altura N. Pej, Si el número es 5 se obtiene:

```
    *
   * *
  * * *
 * * * *
* * * * *
```

44. Igual que en el ejercicio anterior anterior pero además, ahora el usuario introduce el carácter a dibujar:

```
Tamaño? 5
Caracter? e
    e
   e e
  e e e
 e e e e
e e e e e
```

45. Adivina el número:

Crea un programa que simule el juego de adivinar un número. El funcionamiento del juego es el siguiente:

- El programa "piensa" un número entero aleatorio entre 1 y 100.
- A continuación pide un número entero al usuario.
- Mientras el número del usuario no sea el número "pensado", el juego no acaba.
- Si el número del usuario es mayor que el "pensado", se debe mostrar el mensaje: *"Te has pasado!!"*
- Si el número del usuario es menor que el "pensado", se debe mostrar el mensaje: *"Te has quedado corto!!"*
- Cuando el usuario meta el mismo número que el "pensado", se muestra el mensaje: *"Lo lograte!"* y el juego acaba.

46. El juego de las bolas:

Este es un juego donde el jugador compite contra la máquina. Para ello:

- El usuario comienza introduciendo un numero natural positivo que representará la cantidad de bolas inicial que hay en una urna.
- A continuación, la maquina decide quien empieza de forma aleatoria.
- En cada turno, el usuario o la máquina podrán retirar 1,2 ó 3 bolas (es decir, eligen restar 1, 2 o 3).
  - Si es el jugador, se le preguntará cuantas bolas quiere sacar (hay que controlar que sean 1, 2 o 3).
  - Si es la máquina, decidirá de forma aleatoria si saca 1,2 o 3 bolas.
- Los turnos se repiten de forma alternativa hasta que uno de los dos retira la última bola. Quien haga eso, pierde.

Mejora: haz que la máquina decida si quita 1, 2 o 3 bolas en su turno dependiendo de las bolas que haya en la urna (y no de forma aleatoria).

Programación Orientada a Objetos Básica

47.- Se desea implementar en JAVA la clase **Coche**. Cada coche debe tener un color, una anchura, una altura, un numero de puertas y un numero de ruedas.

a) Programa la clase coche usando un constructor con parámetros. Ten en cuenta que todo coche tiene 4 ruedas siempre.
b) Implementa el método *arrancar()* para arrancar el coche. Un coche sólo puede arrancarse si está parado/apagado. El método mostrará por pantalla si consigue arrancar el coche o no.
c) Implementa el método *parar()* para parar/apagar el coche. Un coche sólo puede pararse si está arrancado. El método mostrará por pantalla si consigue parar el coche o no.
d) Implementa el método *desplazarse()* para hacer que el coche se mueva. Un coche sólo puede moverse si está arrancado. El método mostrará por pantalla si consigue desplazarse con el coche o no.
e) Añade el atributo de clase *gasolina* (tipo entero), el cual indicará la cantidad de gasolina que posee el coche en cada momento (valores entre 0 y 50).
f) Haz que el método arrancar consuma 5 unidades de gasolina y el método desplazarse, 20 unidades.
g) Crea el método echar gasolina, el cual añadirá tantas unidades de gasolina como indique su argumento. Recuerda que para echar gasolina debes apagar el coche.
h) Crea el método *chequear()* el cual devolverá un valor booleano. Verdadero si el coche tiene gasolina y falso en caso contrario. Además, el coche sólo podrá arrancar si este método devuelve true.
i) Crea la el método *pintar()* el cual "pintará" el coche con el color que se le pase como parámetro.
j) Implementa el método *toString()* para la clase coche.

48.- Crear la clase **Punto**, la cual tendrá dos atributos: X e Y. También son necesarios los siguientes métodos:

- `moverHorizontal`: recibe un numero entero con signo y desplaza horizontalmente el punto con ese valor.
- `moverVertical`: igual que el anterior pero verticalmente.
- `mostrar`: muestra el punto de la siguiente forma: *Punto (X,Y)*

49.- Crea la clase **Cafetera** con los atributos *capacidadMaxima* y *cantidadActual*. También se necesitan los siguientes métodos:

- El constructor debe indicar la capacidad máxima en 1000 cc y la capacidad actual en 0 (cafetera vacía).
- `echarCafe`: recibe una cantidad de café y lo anota en la capacidad actual. Hay que controlar que la cantidad recibida no sea mayor que la capacidad máxima (si es así, la cafetera se llena completamente y se desecha el sobrante)
- `llenarCafetera`: pone la cafetera a su valor máximo de cantidad.
- `VaciarCafetera`: tira todo el café de la cafetera.
- `servirTaza`: recibe un numero que indica la capacidad de la taza y hace que la cafetera pierda esa cantidad de café. Hay que controlar si la cafetera puede servir la taza.
- `cantidadCafe`: muestra cuanto café queda en la cafetera: *Queda xxxxxx cc de café en la cafetera.*

50.- Crear una clase llamada **Hora**, con los atributos horas, minutos y segundos. Dicha clase podrá ser construida indicando los tres atributos, solo las horas y los minutos o sólo las horas. La clase tendrá los métodos necesarios para modificar la hora en cualquier momento de forma manual y para mostrar la hora que tiene en un momento dado. Hay que mantener los atributos con valores correctos en todo momento.

51.- Crea la clase **Rectángulo**, la cual almacenará la base y la altura. Estos atributos serán privados y no deben poder accederse desde el exterior. Además, la clase tendrá los métodos necesarios para calcular el área (*base * altura*), el perímetro (*2*(b+h)*) del rectángulo y un método para representar en texto la clase:

```
----------------------------------------
Rectángulo con base X y altura Y
Área: A
Perímetro: P
----------------------------------------
```

52.- Crea una clase **Aritmética**, la cual tendrá dos valores numéricos decimales como propiedades de clase y los métodos necesarios para calcular (que no mostrar), la suma, resta, multiplicación, división y potencia de esos valores.

53.- Crea la clase **Persona** con los siguientes datos: Una persona debe tener nombre (cadena), edad (entero mayor a 0), sexo ('M' o 'F') y país (España, Bulgaria, Colombia o Venezuela).

Además tendrá dos métodos:

- `saludar`: el cual debe devolver (que no mostrar) un saludo de la forma: *"Hola! Me llamo nombre y soy de pais"*
- `mostrarDatos`: muestra de forma ordenada los datos de la persona.
- `EsMayorEdad()`: devuelve un booleano que indica si la persona es mayor de edad o no.
- `toString()`: método para mostrar la información de una persona.

b) Mejora el método saludar para que reciba un número entero entre 1 y 3 como parámetro. Dicho número indica el saludo que se debe realizar. Es decir, ahora el método tendrá tres saludos distintos y el parámetro de entrada decidirá cual devuelve.

54.- Queremos crear un juego en el que vamos a tener unas criaturas llamadas *"Monstros"*. Para ello hay que crear una clase que represente estas criaturas.

De cada *Monstro* necesitamos guardar su nombre, sus puntos de esfuerzo, sus vidas totales, sus puntos de hambre y su estado (vivo o muerto). Además, cada *Monstro* puede comer, comunicarse, jugar y dormir.

De momento, las reglas que rigen el juego son:

- El constructor sólo necesita el nombre del *Monstro*.
- Un *Monstro* empieza con 10 puntos de esfuerzo, 3 puntos de hambre, 7 vidas y, obviamente, VIVO.
- Los valores de esfuerzo, hambre y vidas no pueden superar los valores iniciales.
- Al jugar gasta 3 puntos de esfuerzo y 1 de hambre.
- Al comunicarse gasta 1 punto de esfuerzo.
- Si come, gasta dos puntos de esfuerzo y gana dos de hambre. Pero si además come su comida favorita, ganará 3 puntos de hambre.
- Al dormir gana 5 puntos de esfuerzo.
- Si los puntos de esfuerzo o de hambre llegan a cero, el *Monstro* perderá una vida y volverá a vivir con los valores por defecto.
- Si el *Monstro* pierde todas las vidas, muere, todos sus atributos tomarán el valor -1 y no podrá realizar ninguna acción .

55.- Crear una clase **Libro** que contenga los siguientes atributos: ISBN, Titulo, Autor, Número de páginas (todos obligatorios y privados).

Crear sus respectivos métodos get y set correspondientes para cada atributo. Crear el método `toString()` para mostrar la información relativa al libro con el siguiente formato: *"El libro TITULO con ISBN creado por AUTOR tiene NPAGINAS"*

Para probar la clase, crear 2 objetos Libro (los valores que se quieran) y mostrarlos por pantalla. Por último, indicar cuál de los 2 tiene más páginas.

56.- Crea una clase llamada **Cuenta** que tendrá los siguientes atributos privados: titular y cantidad. Al crear la clase, el titular será obligatorio pero la cantidad puede ser opcional.

Ademas de los métodos getters y setters, la clase tendrá los siguientes:

- `ingresar`: se ingresa una cantidad a la cuenta, si la cantidad introducida es negativa, no se hará nada.
- `retirar`: se retira una cantidad a la cuenta, si restando la cantidad actual a la que nos pasan es negativa, la cantidad de la cuenta pasa a ser 0.
- `toString`: muestra una cadena con los datos de la cuenta: *Esta cuenta pertenece a xxxxx y posee actualmente una cantidad de yyyy €.*

57.- El objetivo de este ejercicio es crear la clase **Urna** cuyos objetos pueden contener bolas blancas y bolas negras y, además, nos va a permitir realizar operaciones básicas sobre ellas. Posteriormente crearemos un test para ver si la clase funciona correctamente.

- La clase tendrá un par de variables de instancia privadas en las que se almacenará el número de bolas de cada color.
- También tendrá un constructor que permita crear instancias de la clase con un número inicial de bolas blancas y negras pasadas como parámetros.
- Además incluirá métodos para:
  - Consultar el número total de bolas.
  - Extraer una bola y saber su color. El color vendrá dado simplemente por un carácter 'n' o 'b'. Para extraer una bola aleatoriamente se suma el número de bolas blancas y negras y se toma un número aleatorio entre 1 y esa suma. Si ese número es menor o igual que el número de bolas blancas, suponemos que la bola que sale es blanca. En otro caso, suponemos que la bola es negra.
  - Introducir una bola de un color determinado.

A continuación, hay que crear el archivo `TestUrna.java` e implementar el test de la siguiente forma:

- Debe crear una urna con un número de bolas blancas y negras aleatorias (entre 1 y 10).
- Mientras quede más de una bola en la urna, se sacan dos bolas:
  - Si ambas son del mismo color, se mete una bola blanca en la urna.
  - Si son de diferente color, metemos una bola negra.
- Por último, cuando quede una sola bola, se saca y se indica su color,

Hay que analizar si, partiendo del número de bolas iniciales, la última bola que indica el test es correcta. Para ello recomiendo implementar un modo *verbose*, es decir, un modo que vaya indicando qué va pasando.

Ejemplo de ejecución:

```
Urna creada con 4 bolas blancas y 4 bolas negras
Se ha extraido una bola b
Se ha extraido una bola b
Se ha metido una bola BLANCA.
Quedan 7 bolas en la urna.
    3 blancas y 4 negras.
Se ha extraido una bola b
Se ha extraido una bola b
Se ha metido una bola BLANCA.
Quedan 6 bolas en la urna.
    2 blancas y 4 negras.
Se ha extraido una bola b
Se ha extraido una bola n
Se ha metido una bola NEGRA.
Quedan 5 bolas en la urna.
    1 blancas y 4 negras.
Se ha extraido una bola n
Se ha extraido una bola b
Se ha metido una bola NEGRA.
Quedan 4 bolas en la urna.
    0 blancas y 4 negras.
Se ha extraido una bola n
Se ha extraido una bola n
Se ha metido una bola BLANCA.
Quedan 3 bolas en la urna.
    1 blancas y 2 negras.
Se ha extraido una bola n
Se ha extraido una bola b
Se ha metido una bola NEGRA.
Quedan 2 bolas en la urna.
    0 blancas y 2 negras.
Se ha extraido una bola n
Se ha extraido una bola n
Se ha metido una bola BLANCA.
Quedan 1 bolas en la urna.
    1 blancas y 0 negras.
Se ha extraido la bola final de color b
```

Arrays

58.- Sea el array notas = [6,3,9,7,5,8,10,2,4,5]
Crea un programa que muestre por pantalla la media ponderada de las notas.

59.- Sea el array `flags =[0,1,0,-1,0,-1,-1,-1,0,0,-1].` Crea un nuevo array llamado *tivic* que sea copia del anterior.
A continuación recorre el primer array (*flags*) y cambia todos los valores negativos por el valor 8.

60.- Crea un array de 100 elementos dónde cada elemento sea un número aleatorio entre 1 y 100:
a) Hacer un programa que muestre los valores múltiplos de 5.
b) Otro que sólo muestre los números pares.
c) Otro que muestre el mayor de todos los elementos del array y su posición (si hay varios iguales, muestra el primero)

61.- Haz un programa que dados dos arrays (los que tu quieras) diga sin esos arrays son iguales o no. Dos arrays son iguales si sus elementos son iguales y están en la misma posición.

62.- Haz un programa que dado el array `valores = [2,4,6,8,1,3,5,7]`, genere otro array con los mismos valores pero con el valor 200 en la posición 4 y el valor 300 en la posición 7 (y el resto de valores afectados desplazados una posición).

```
Valores: 2 4 6 8 1 3 5 7
Otra: 2 4 6 200 8 1 300 3 5 7
```

63.- Haz un programa que dado un array, pej `valores = [2,4,6,8,1,3,5,7]`, y un numero que se pida por teclado, indique si ese número está en la lista e informe convenientemente.

64.- Usando arrays: escribe un programa que pida 10 números naturales y a continuación, indique cuál es el valor máximo de esa sucesión de números, el número de veces que aparece y sus respectivas posiciones. Ejemplo:

```
7 10 143 10 52 143 72 10 143 7
El valor máximo es el 143
Aparece 3 veces en las posiciones 2,5,8
```

65.- Uno de los métodos más famosos para ordenar un array numérico es el *algoritmo de ordenación por selección*. Este algoritmo consiste en:

- Se busca el elemento más pequeño del array se coloca en la primera posición.
- Se hace lo mismo que en el punto anterior pero ahora se toma todo el array menos la primera posición (porque ya está ordenada).
- Se repite el primer punto pero ahora solo se toma todo el array menos las dos primeras posiciones (porque ya están ordenadas).
- Se repite el primer punto hasta que ordeno todos los elementos del array.

50	26	7	9	15	27	Array original
7	26	50	9	15	27	Se coloca el 7 en primera posición. Se intercambian 7 y 50
7	9	50	26	15	27	Se coloca el 9 en segunda posición. Se intercambian 9 y 26
7	9	15	26	50	27	Se coloca el 15 en tercera posición. Se intercambian 15 y 50
7	9	15	26	50	27	El menor de los que quedan es el 26. Se deja en su posición
7	9	15	26	27	50	Se coloca el 27 en quinta posición. Se intercambian 27 y 50

Crea un programa en el que se defina un array de 10 números enteros aleatorios (entre 1 y 100) y ordene ese array usando la ordenación por selección.

66.- Define la clase **Estudiante** de la siguiente forma: De cada estudiante debemos saber su nombre, sus apellidos y una lista de calificaciones que va a tener en el curso. El tamaño de esa lista se pasará en el constructor. Además, todas las propiedades deben ser privadas.
Al comienzo, todas las calificaciones del estudiante tendrán el valor -1 (no presentado).
Define los siguientes métodos de la clase:

- `InsertarNota(double nota)`: meterá la nota que se le pasa como parámetro en la lista de notas. Devolverá true si consigue meterla o false si no consigue hacerlo (porque la lista ya esté llena).
- `MostrarNotas()`: devolverá una cadena bien maquetada con las notas de ese estudiante.
- `CalcularMedia()`: devolverá la media de las notas que tenga ese estudiante hasta el momento. Los 'no presentados' no deben contarse para calcular la media.
- `IndicarRangos()`: este método contará cuantos insuficientes, suficientes, notables y sobresalientes tiene el estudiante y devolverá esos valores.
- `ToString()`: Este método debe generar la información del estudiante. Para ello sigue la siguiente plantilla:

```
Estudiante: nombre y apellidos
Notas: lista de notas
Rangos: numero de  insuficientes, suficientes, notables
y sobresalientes.
```

```
Media: media
```

67.- Implementa la clase lista de cadenas. Dicha clase va a servirnos para facilitar el uso de arrays a gente que está aprendiendo a programar.

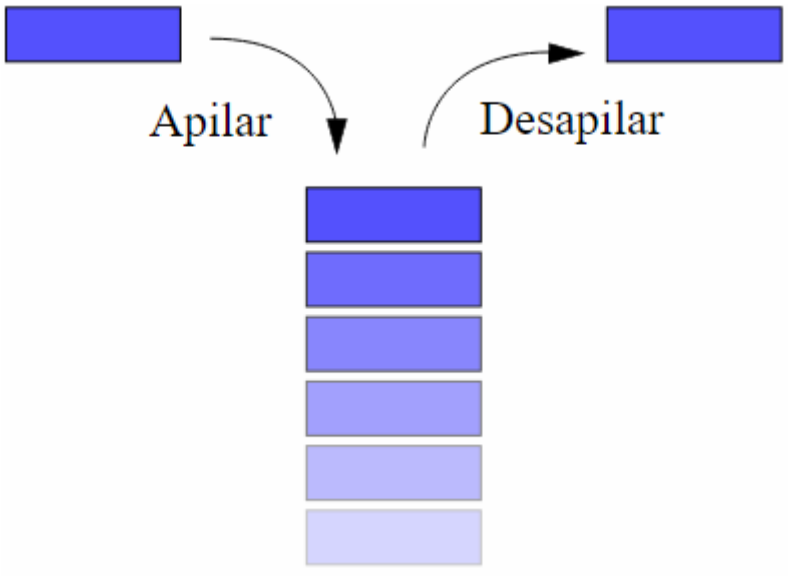
La clase va a tener la propiedad lista, la cual será un array del tamaño indicado en el constructor, por tanto, el constructor de la clase sólo necesita un parámetro: el tamaño de la lista.
También tendrá un número encargado de indicar la siguiente posición vacía de la lista.

La lista, al principio, estará rellena de cadenas vacías.

Los métodos de la clase serán los siguientes:

- `aniadir(String elemento)`: añade el elemento que se le pasa como argumento en la siguiente posición vacía de la lista. Sino puedo hacerlo, lo
- `aniadir(String elemento, int pos)`: añade el elemento que se pasa como primer argumento en la posición indicada en el segundo argumento. Si ya hay algo en esa posición, lo machaca. Si la posición indicada no existe, debe indicarse
- `tamanio()`: devuelve el tamaño de la lista.
- `vacia()`: devuelve un booleano indicando si la lista está vacía o no.
- `obtener(int pos)`: devuelve el elemento que hay en la posición indicada como argumento. Si la posición indicada no existe, debe indicarse.
- `buscar(String elemento)`: devuelve la posición del elemento que se pasa como argumento en la lista. Si el elemento no está, devuelve -1.
- `borrar()`: borra todos los elementos de la lista (los pone como cadena vacía).
- `borrar(int pos)`: borra el contenido de la posición indicada como argumento (es decir, coloca la cadena vacía). Si la posición indicada no existe, debe indicarse.
- `borrar(String elemento)`: borra el contenido de la primera celda que encuentre con el mismo contenido que el argumento (es decir, coloca la cadena vacía). Si no puede borrar nada, lo indica.
- `trozo(int pos1, int pos2)`: devuelve un trozo de la lista (un array) tomando como inicio el argumento de menor valor y como fin el argumento de mayor valor. Si alguno de los argumentos no es correcto (se sale del tamaño d ella lista), devolverá un array vacío.

68.- Una pila es una estructura informática que sirve para almacenar/recuperar datos y cuyo funcionamiento se basa en que el primer elemento que se almacena, es el último en poder obtenerse.



No puedo "sacar" un elemento si no se han extraído los que tiene encima. Es decir, si no está en la cima de la pila.

El objetivo de este ejercicio es crear una clase que imite el funcionamiento de la estructura tipo Pila de números. Para ello la clase va a tener estas propiedades:

1. Un array de números enteros que usaremos para almacenar los elementos.
2. Un número encargado de indicarnos cual el índice del array que hace de cima de la pila.

El constructor sólo debe tener como argumento el tamaño que va a tener la pila. Además, cuando se crea la pila por primera vez, esta debe estar vacía.

Los métodos que debe tener la clase son:

- `insertar(int elemento)`: si se puede, añade el elemento a la pila. Sino, informa de que no ha podido.
- `cima()`: devuelve el elemento que está en la cima (pero no lo elimina).
- `extraer()`: devuelve el elemento que está en la cima y lo saca de la pila (lo elimina).
- `vacia()`: devuelve un booleano indicando si la pila está vacía o no.
- `llena()`: devuelve un booleano indicando si la pila está llena o no.
- `toString()`: representa la pila en su estado actual.

Para probar el funcionamiento correcto de la clase, adjunto el fichero `TestPila.py`

Relaciones de Agregación y Composición

69.- El objetivo es implementar las **clases punto, círculo y cilindro** según se indica más adelante y probarlas usando el archivo `TestFiguras.java` que se adjunta.
La clase punto tendrá dos propiedades de tipo número entero que se encargarán de almacenar la posición X y la posición Y del punto en el espacio. Además tendrá los siguientes métodos:

- El constructor, el cual necesitará dos valores: uno para la X y otro para la Y.
- Un método llamado `toString()` que devolverá una cadena representando el punto (indicando sus posiciones).
- Un método llamado `trasladar(a,b)`, encargado de trasladar el punto tantas unidades como indican los parámetros que se les pasa (es decir, sumar esos valores a las posiciones respectivas del punto).

La clase círculo tendrá como propiedades el radio (un número) y el centro (un punto). Sus métodos serán:

- Un constructor que se le pasará como argumento un punto y el valor del radio.
- Un método para calcular el área del círculo.
- Un método para calcular el perímetro del círculo.
- El método `toString()` que devolverá una cadena representando el circulo (indicando su punto central y su radio).
- El método `trasladar(a, b)`, encargado de trasladar el punto central del círculo.

Por último, para la clase Cilindro hace falta una altura (número) y un círculo que hará de base. Además sus métodos serán:

- El constructor cuyos parámetros serán la base (un circulo) y la altura del cilindro.
- Método para calcular el volumen del cilindro.
- El método `toString()` que devolverá una cadena representando el cilindro (indicando su base y su altura).

NO hay que usar herencia, hay que usar composición (utilizar unas clases dentro de otras).

70.- La clase **Guerrero** será una clase encargada de almacenar la información de un posible guerrero/a, cual tendrá como atributos privados:

- `nombre`: nombre del guerrero/a.
- `Salud`: un número que indica la salud máxima del guerrero.
- `Ataque`: número que indica su fuerza de ataque.
- `Escudo`: número que indica su resistencia a los golpes.

Además tendrá otro atributo privado que sirva para controlar la salud actual del guerrero. Cuando se crea un guerrero por primera vez, su salud actual coincide con la salud máxima de este.

La única acción que puede hacer nuestro guerrero es `atacar`. Dicha acción devuelve la fuerza del ataque del guerrero. Define también el método `toString()`

Por otro lado, la clase **arma**. Va a representar un arma que podrá ser usada por un guerrero. Un arma va a tener:

- `nombre`: nombre del arma.
- `resistencia`: un número que indica el aguante del arma.
- `poder`: número que indica la cantidad de daño que hace el arma.

Todos los atributos serán privados. El único método que tendrá la clase arma es `toString()`

Se pide:

a) Implementa ambas clases y crea un par de objetos de cada una de ellas.

b) Añade la propiedad '`muerto`' al guerrero que servirá para indicar si el guerrero está vivo o muerto. Un guerrero muerto no podrá atacar ni realizar ninguna acción.

c) Modifica la clase guerrero para que ahora todo guerrero tenga un arma (el arma debe existir previamente) y mejora el método `atacar` para que también tenga en cuenta el poder del arma.

d) Mejora el método `atacar` del apartado anterior para que reciba como argumento a otro guerrero que ya exista (será el guerrero que defiende). Ahora, atacar debe quitar al guerrero que defiende una cantidad de salud igual al ataque total del guerrero que ataca menos el escudo del guerrero que defiende. Además debe verificar si el guerrero que defiende queda vivo o muerto tras el ataque.

e) Mejora el método `atacar` para que cada vez que un guerrero ataque, su arma pierda un punto de resistencia. Si un arma se queda sin resistencia no se podrá atacar con ella (no se tomará en cuenta su poder al atacar).

f) La clase arma ahora va a tener un atributo privado llamado `nivel`. Cuando se crea un arma su nivel empieza en 1. También debe tener un método llamado subir nivel: este método hace que el nivel del arma aumente y, en consecuencia, su resistencia aumente en 2 puntos y su ataque en 1.

g) El guerrero ahora tendrá un método llamado `mejorarArma` el cual recibirá el nombre del arma a mejorar. Si el nombre del arma a mejorar coincide con el arma que lleva el guerrero, el arma subirá un nivel.

h) Nuestro guerrero ahora va a tener tres atributos más: `nivel`, el cual indicará el nivel del guerrero (un número entero comenzando por 1), `experienciaActual`, la indicará qué cantidad de experiencia tiene el guerrero en ese momento (otro número entero que debe comenzar en 0 en cero) y `siguienteNivel`, un número que se calculará usando la formula: `nivel*10+nivel*1.5` (si salen decimales, se redondea hacia arriba).

i) Crea el método `subir`, el cual devuelve True si el guerrero puede subir de nivel o False en caso contrario. Un guerrero subirá de nivel si sus puntos de experiencia actuales coinciden con el valor de `siguienteNivel`.

j) Implementa el método `subidaNivel` en el guerrero, el cual aumentará en 1 el `nivel` del guerrero, pondrá la `experienciaActual` a 0, volverá a calcular `siguienteNivel`, sumará 1 punto al `ataque` y al `escudo` del guerrero y 10 puntos a su `salud máxima`.

k) Modifica el método atacar para que cada vez que el guerrero ataque, el guerrero gane 3 puntos de experiencia si usa un arma o 5 su no usa arma o está rota. Además hay que comprobar si el guerrero sube de nivel tras el ataque realizado.

71.- Vamos a hacer una baraja de cartas francesa usando POO, la cual nos servirá para programar juegos más adelante.
Para empezar hay que definir la **clase carta**. Una carta tiene un número entre 1 y 12 y un palo, el cual será un valor de los siguientes: Picas, Tréboles, Corazones o Diamantes.

Una carta también va a tener una forma que irá en función de su palo. Para representar el palo de una carta usaremos los siguientes códigos UNICODE:

- Pica: `"\u2660"`
- Trébol: `"\u2663"`
- Corazón: `"\u2665"`
- Diamante: `"\u2666"`

El único método de la clase carta será `toString` el cual devolverá una representación de la forma: `[numero - forma]`

Una baraja va a estar compuesta por un conjunto 48 cartas: los 4 palos indicados en las cartas y 12 cartas por palo.
La **clase baraja** no necesita ningún parámetro para ser creada, pero cuando se instancie un objeto de ella, debe crearse un mazo[1] el cual tendrá cada una de las cartas indicadas.

Las operaciones que podrá realizar la baraja son:

- `mostrar:` muestra el mazo de la baraja tal y como está en ese momento. Intenta que la representación sea lo más clara posible (no mostrar todas las cartas una debajo de otra) Intenta que se vea todo el mazo a la vez, pej: mostrar 4 columnas de 12 cartas cada una.
- `barajar:` cambia de posición todas las cartas aleatoriamente.

[1] Todas las cartas del mazo van en una variable (tu decides la estructura). No es posible usar más de una variable para implementar el mazo.

72.- Mejora la clase baraja del ejercicio anterior añadiendo las siguientes funcionalidades:

Ahora la clase debe tener una pila de cartas usadas dónde irán las cartas que se sacan del mazo y los siguientes métodos:

- `cartasDisponibles:` indica el número de cartas que aún se pueden repartir.
- `siguienteCarta`: devuelve primera carta del mazo y la elimina (va a la pila de usadas). Si no hay más cartas en el mazo, se indica al usuario que no hay más cartas.

- `darCartas`: dado un número de cartas que nos pidan, le devolveremos ese número de cartas (y van a la pila de usadas). En caso de que haya menos cartas que las que se piden, debemos indicárselo al usuario y no se devuelve ninguna.
- `cartasUsadas`: muestra aquellas cartas que ya han salido, si no ha salido ninguna, se indica al usuario.
- `reiniciar`: junta las cartas de la pila de usadas con las cartas del mazo. La pila debe quedarse vacía y el mazo con todas las cartas de la baraja

73.- Haciendo uso de la baraja implementada en el ejercicio anterior, vamos a definir **el juego** de la carta mas alta para varios jugadores.
Para ello hay que crear **la clase jugador**, la cual tendrá como único parámetro en el constructor el nombre. Aparte, de cada jugador es necesario saber la carta que tiene en un momento dado de la partida y el numero de victorias que lleva.

El juego comienza creando **la baraja** de cartas que se va a usar y preguntando cuantos jugadores van a jugar. Si el numero de jugadores es mayor al numero de cartas de la baraja, no se podrá jugar (obviamente) y el juego finalizará.

En cualquier otro caso, se crearan tantos jugadores como se han indicado comenzará el juego. El juego se compone de varias rondas. En cada ronda se realizan las siguientes acciones:

- Se mezcla la baraja.
- Si se puede, se reparte una carta a cada jugador (siguiendo el orden con el que se han apuntado al juego).
- Se miran todas las cartas y se decide la carta ganadora.
- Se anota la victoria en la propiedad correspondiente del jugador ganador:
  - Primero se comparan los números. Gana el de mayor valor.
  - En caso de empate, se miran los palos. El orden de mayor a menor valor sería: Corazones, Picas, Diamantes y Tréboles
- Se “devuelven” las cartas al mazo.

El juego acaba cuando no se puede repartir cartas a todos los jugadores.
Al salir del juego debe indicarse cuál es el jugador ganador (el que más victorias tiene)

74.- Se desea implementar el juego de la ruleta rusa en POO.
Como muchos sabéis, se trata de un número de jugadores dónde se usa un revolver con un sola bala en un tambor de N huecos. En cada ronda los jugadores van disparándose en la cabeza. Pierde el jugador que muere...obviamente.

Las clases que necesita el juego son:

Clase Revolver:

El revolver tendrá un constructor que necesitará el número de huecos que posee el tambor. Posteriormente creará una estructura para simular ese tambor (crear una estructura de N huecos y colocar la bala en uno de ellos de forma aleatoria) . Aparte, son necesarios los siguientes atributos:

- posición actual: indica posición del tambor donde va a golpear el martillo del revolver si se dispara (en esa posición puede que esté la bala o no).
- posición bala: indica la posición del tambor donde se encuentra la bala.

Estas dos posiciones, se generaran aleatoriamente.

Las acciones que se pueden realizar con el revolver son:

- `disparar()`: devuelve `true` si la bala coincide con la posición actual.
- `moverTambor()`: cambia a la siguiente posición del tambor.
- `toString()`: muestra información del revolver (posición actual del tambor y donde está la bala)

A tener en cuenta:
Cuando se dispara el revolver, el tambor avanza y coloca un nuevo hueco en el martillo (listo para ser disparado si hay bala). Además, el tambor nunca deja de girar (es decir, cuando llega al final, vuelve al principio).

Clase Jugador:

Del jugador se necesita un nombre y una variable que indique si está vivo o muerto.
Además, un jugador puede `disparar` un revolver: el jugador se apunta y se dispara, si la bala del revolver se dispara, el jugador muere.

Clase Juego:

Esta clase será la encargada de gestionar todo el juego. Para ello se necesitará el revolver que se va a usar y el número de jugadores que van a jugar, el cual no puede ser mayor al tamaño del tambor del revolver usado.
A continuación, se crearán tantos jugadores como se han indicado y se almacenaran en una variable de clase (usando la estructura que creas conveniente).

Los métodos que tendrá esta clase van a ser:

- `ronda()`: cada jugador toma el revolver y se dispara. Se informará qué ha pasado con cada jugador después de cada disparo.
- `finJuego()`: devuelve True si hay algún jugador muerto.

Implementa el juego de las siguientes formas:

a) Se disparan todos los jugadores y luego se comprueba si ha muerto alguno para ver si el juego acaba.
b) Se van disparando los jugadores por orden y en el momento que uno muera, el juego acaba.

Herencia Simple

75.- Implementa la clase **Empleado** la cual va a tener el atributo privado nombre. Aparte del constructor, crea un método para obtener el nombre y el `toString`, el cual devolverá la cadena: "`Soy el empleado` *`nombre`*".

a) Crea las clases **Operario** la cual hereda de Empleado. Un operario tendrá como propiedad privada un número entero que indicará su código de trabajador. Hay que crear métodos para obtener/cambiar ese atributo. Además, hay que sobreescribir el método `toString` para que devuelva la cadena: "`Soy el operario nombre con el código código`"

b) Crea la clase **Directivo** que heredará de Empleado. En esta clase tan solo hay que sobreescribir el método `toString` para que devuelva la cadena: "`Soy` *`nombre`*`, directivo de la empresa`"

c) Crea las clases **Oficial** y **Técnico**, las cuales heredarán de Operario. Para esas clases tan solo hay que sobreescribir el método `toString` de modo que se devuelvan las cadenas:
   - Para oficial: "`Soy el oficial nombre con el código código`".
   - Para Técnico: "`Soy el técnico nombre con el código código`".

d) Prueba las clases creadas con el siguiente código:

```
Empleado E1 = new Empleado("Jaime");
Directivo D1 = new Directivo("Marcelo");
Operario OP1 = new Operario("Jose",34);
Oficial OF1 = new Oficial("Dani");
Tecnico T1 = new Tecnico("Carlos");
System.out.println(E1);
System.out.println(D1);
System.out.println(OP1);
System.out.println(OF1);
System.out.println(T1);
```

76.- Crea la clase **Mamífero**, el cual tendrá como propiedades numero de patas y media de vida y como comportamientos comunicarse y dormir. Comunicarse devolverá una cadena vacía y dormir mostrará un mensaje con el numero de horas que suele dormir un mamífero (8). Implementa también el método `toString`.

Ahora crea la clase **Perro** y la clase **Gato**. Ambas clases deben tener `raza` y `nombre` como propiedad, las cuales serán requeridas como parámetro en el constructor.

La clase Perro tendrá además la propiedad `mezcla`, la cual indica si el perro es de pura raza o no. Aunque suene raro, todos los perros al nacer son de pura raza.
Los métodos de la clase perro serán los heredados adaptados a un perro: se comunica con ladridos (devuelve un ladrido) y duerme unas 18 horas. Además tendrá el método `jugar` el cual mostrará la cadena "*Nombre* `está jugando`".
Implementa también get y set para la propiedad `mezcla` y sobreescribe el método `toString`

Por su parte Gato, aparte de un nombre, tendrá la propiedad `pelaje` con los siguientes valores posibles: corto, largo o semilargo. Al nacer todos los gatos tienen un pelaje corto. Crea también get y set para esta propiedad.
Un gato maulla si quiere comunicarse y suele dormir en torno a 15 horas al día. Además a los gatos les gusta `cazar`. Crea ese comportamiento de modo que se muestre por pantalla "Mi gato de *numpatas* patas está cazando". No olvides también corregir el método `toString`.

Implementa todas las clases indicadas haciendo uso de la herencia simple y crea un pequeño programa para probarlas. (Pej: crear un par de perros y gatos y ver sus características).

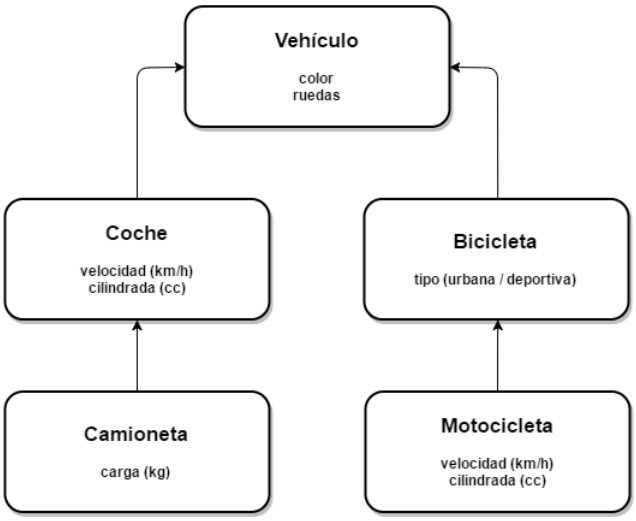
77.- Vamos a crear un gestor de archivos multimedia usando la POO.
Para ello se debe implementar la **clase Multimedia** la cual tendrá título, formato y duración como atributos (se le pasan como parámetros en el constructor).
El formato será uno de los siguientes: mp3, wav, aac, mp4, mkv, mov o flv. Hay que comprobar que se le pasa uno de esos valores, si no es así, se elegirá el formato mp3.

Esta clase tendrá además, un método para devolver cada atributo, el método `toString` y un método llamado `igual` que recibirá otro objeto multimedia y devolverá True o False si son iguales o no. Dos objetos son iguales si su formato y su duración son iguales.

Implementa una **clase Película** que herede de Multimedia y que tenga, además de lo heredado, un actor principal y una actriz principal (privados ambos).
Tendrá, además, métodos para obtener los nuevos atributos, el método `toString` sobreescrito para que muestre la información de los nuevos atributos y deberá "corregir" el método `igual` para que tenga en cuenta al actor y a la actriz.

Por otro lado tendremos la **clase Canción** que también hereda de Multimedia y posee los atributos Artista y Editor (ambos son cadenas) y género (pop, rock, flamenco, hiphop, R&B, o reggaeton). Todos privados.
Se deben realizar los métodos para obtener el valor de los nuevos atributos y el método `toString` corregido para que muestre la información de los nuevos atributos.

78.- Vamos a implementar el siguiente diagrama:



Tan sólo es necesario implementar de cada clase: el constructor (cuyos argumentos serán los atributos indicados) y el método `toString` (indicando en cada caso de qué tipo es y cuales son sus atributos y los valores que poseen). Además todos los atributos van a ser protegidos.

Para probar el correcto funcionamiento, implementa un programa que realice los siguientes pasos:

1. Crea un par de objetos de cada clase y añádelos a una lista de vehículos.
2. Recorre la lista de vehículos y muestra la información de cada objeto almacenado
3. Pregunta al usuario por un número. Ese numero indicará el número de ruedas de un vehículo.
4. Usando el número anterior, recorre la lista de vehículos y muestra sólo aquellos que coincidan en número de ruedas. También debes mostrar un mensaje del tipo: `Se han encontrado {} vehículos con {} ruedas.`

79.- Queremos desarrollar un programa usando POO que permita la gestión de una empresa agroalimentaria que trabaja con tres tipos de productos: productos frescos, productos refrigerados y productos congelados.

Todos los productos llevan la siguiente información común: fecha de caducidad y número de lote. A su vez, cada tipo de producto lleva alguna información específica:

- Los productos frescos deben llevar la fecha de envasado, el país de origen y el método de conservación usado *(una cadena)*.
- Los productos refrigerados deben llevar el código del organismo de supervisión alimentaria *(un número entero)*, la fecha de envasado, la temperatura de mantenimiento recomendada *(un double)* y el país de origen.
- Los productos congelados deben llevar la fecha de envasado, el país de origen y la temperatura de mantenimiento recomendada.

Hay tres tipos de productos congelados: congelados por aire, congelados por agua y congelados por nitrógeno.

- Los productos congelados por aire deben llevar la información de la composición del aire con que fue congelado: % de nitrógeno, % de oxígeno, % de dióxido de carbono y % de vapor de agua *(todos tipo double).*
- Los productos congelados por agua deben llevar la información de la salinidad del agua con que se realizó la congelación en gramos de sal por litro de agua *(tipo double)*.
- Los productos congelados por nitrógeno deben llevar la información del método de congelación empleado *(una cadena)* y del tiempo de exposición al nitrógeno expresada en segundos *(un numero entero)*.

Crea las clases indicadas usando una relación de herencia y teniendo en cuenta que cada clase debe disponer de constructor, de un método para recuperar el valor de sus atributos y tener un método que permita mostrar la información del objeto cuando sea procedente.

80.- Necesitamos crear la clase **Asignatura**, la cual tendrá un código de asignatura y un número de horas que se imparte a la semana. Además es necesario implementar su método `toString`

Un/a **profesor/a** tendrá un nombre, una lista de asignaturas que puede impartir y el número de horas que está en clase. Su constructor necesita el nombre del profesor y el número de asignaturas que puede impartir. También hay que controlar si el profesor está dando clase o no.

Cuando se crea un profesor ocurre lo siguiente:

- Se van a crear con él un numero de asignaturas aleatorio entre 1 y el numero máximo de asignaturas que puede impartir (OJO por que eso quiere decir que el array puede tener huecos libres)
- Ese número de asignaturas se crearan de forma aleatoria: su código con valores entre 100 y 200 y las horas con valores entre 3 y 8.

- El valor de la propiedad 'horas totales' se obtiene sacando la suma de las horas de cada asignatura que posee (es decir, se rellena con la suma de todas las horas de las asignaturas que posee).
- Obviamente, al crearse un profesor, este no está dando clase.

El nombre y el número de horas serán de ámbito privado. El resto serán protegidas.

El profesor tendrá dos comportamientos:

- `darClase`: controla que el profesor está dando clase. Además, toma una asignatura de las que tiene forma aleatoria y muestra por pantalla: '*Nombre' está dando clase de 'Asignatura'*. Obviamente, si ya se está dando clase, no se puede dar otra clase.
- `parar`: el profesor deja de dar clase e indica por pantalla: '*Nombre' ha finalizado de dar clase*. Sólo se podrá dejar de dar clase si se está dando.
- El método `toString` mostrará todos los datos del profesor, incluidas las asignaturas que tiene y sus horas, las horas totales de clase y si está en clase o no.

De la clase profesor clase **Profesor de Universidad** (que hereda de Profesor). Para dicho tipo de profesor hay que saber si puede investigar o no y el número de horas que investiga. También hay que controlar si el profesor está investigando en un momento dado o no.

A la hora de crear este tipo de profesor hay que tener en cuenta lo siguiente: Debe tener al menos 8 horas totales de clase a la semana. Si esto no ocurre, no puede investigar y sus horas de investigación son 0.

Esta clase necesitará los métodos para realizar las siguientes acciones:

- Ahora el profesor puede también `investigar`. Este método tan solo mostrará por pantalla: *"'nombre' está investigando"*. CUIDADO:
  - Si se está investigando, no se puede dar clase.
  - Si se está dando clase, no se puede investigar.
- Si puede investigar, también puede `parar de investigar`.
- Si el profesor quiere, puede `coger otra asignatura` para dar clase. Esto podrá hacerse si tiene hueco en la lista de asignaturas y siempre y cuando no supere las 25 horas semanales. Si se añade la asignatura, hay que recalcular su numero de horas. Sino, se muestra un mensaje de error al usuario.
- El profesor puede optar a `abandonar asignaturas` que posee. Para ello siempre se eliminará la última asignatura que tenga y habrá recalcular el número de horas de clase del profesor.

El método `toString` mostrará ahora si, en ese momento, está investigando, dando clase o ninguna de las dos.

Clases Abstractas

81.- Revisa los ejercicios 76,77 y 78 de la parte de Herencia y medita qué clases de esos ejercicios deberían ser abstractas. Corrige el código en base a lo debatido en clase.

82.- Crea la clase Abstracta Robot, la cual tendrá como propiedades de clase: aguante, velocidad y potencia. Además tendrá dos métodos abstractos:

- `combatir`: recibe como parámetro un objeto robot que será atacado por el objeto que invoca el método.
- `mostrar`: muestra toda la información del robot en ese momento.

Implementa la clase Robot de Combate, la cual heredará de Robot y tendrá como atributos (además de los heredados): grosor de armadura, estado de armadura y arma.

Define los métodos que crees necesarios en esta clase para implementar correctamente los métodos abstractos de la clase padre. Debes tener en cuenta que:

- El método `combatir` debe decidir qué robot gana el combate y qué daños sufre cada robot.

El método `mostrar` debe mostrar toda la información del robot de combate.

83.- Una conocida web de venta de productos informáticos ofrece una serie de descuentos especiales a sus clientes. Para ello ha dividido a sus clientes en base a la antigüedad que tienen y ha obtenido la siguiente clasificación:

- Clientes tipo A: poseen un 15% de descuento por cada 75€ de compra.
- Clientes tipo B: poseen un 8% de descuento por cada 75€ de compra.
- Clientes tipo C: poseen un 5% de descuento por cada 75€ de compra.

Todo cliente va a tener un nombre y una clave de cliente (cadena alfanumérica de 5 caracteres). Aparte del método `toString`, cada cliente va a tener un método abstracto llamado `calcularDescuento` que recibirá la cantidad de dinero que se ha gastado el cliente en una compra y devolverá el total a pagar por el cliente en base a su descuento.

a) Implementa una jerarquía de clases apropiada para controlar los clientes de cada tipo.

b) Mejora el ejercicio con lo siguiente:

Ahora cada cliente debe tener ademas un historial de las compras que realiza y de lo que ha pagado por cada una de ellas.
Cada compra va a tener *la fecha de la compra* *(Nota: Lo ideal es usar un objeto tipo DATE. Sino, usa una String)*, una lista de artículos comprados *(ver más adelante la clase artículo)*, el

precio total a pagar y si se le puede hacer descuento o no. Es necesario un método llamado `imprimirTicket`, el cual sacará un ticket con todos los datos de la compra en cuestión.

Los objetos de la clase artículo van a tener el nombre del artículo y su precio (número decimal). Ambas propiedades serán privadas. Será necesario también poder mostrar el articulo correctamente.

Finalmente, ahora cada cliente debe poder realizar lo siguiente:

- Mostrar su historial de compras: sacará todas las compras realizadas por ese cliente.
- Calcular dinero total gastado: devuelve la cantidad total gastada por el cliente (incluyendo descuentos).

84.- De una persona necesitamos saber su nombre, sus apellidos, su edad y si está casada o no.
Un profesor es una persona que además trabaja un número de horas a la semana en un centro educativo, tiene una o varias asignaturas* y gana un sueldo. Cuando se contrata a un nuevo profesor, se le asigna su centro de trabajo y nada más.

Todos los profesores van a ser de dos tipos: Titular o Interino.

A un profesor titular se le asignan de forma aleatoria cuatro asignaturas de cualquier grupo, siempre que su total de horas no supere las 22 horas a la semana. Además, su sueldo se calculará teniendo en cuenta lo siguiente: cobran 26.5€ la hora hora y le retienen un 15% de IRPF. Suponemos, además, que todos los meses tienen 4 semanas.

En el caso de un profesor interino, sólo puede dar como máximo dos asignaturas de los dos primeros grupos y sin superar las 10 horas semanales. Su sueldo se calculará teniendo en cuenta que cobran 15€ la hora y se les retiene un 10% de IRPF. Mantenemos la suposición de que todos los meses tienen 4 semanas.

Implementa una jerarquía de herencia válida para las 4 clases de este ejercicio. Aparte del constructor, `toString` y los *getters*, *setters* necesarios, tú decides los demás comportamientos de las clases.

*Las asignaturas disponibles son:

- Asignaturas de ciencias: biología(4), conocimiento del medio(4) y geología(2).
- Asignaturas tecnológicas: tecnología(4), matemáticas(8) e informática(6).
- Asignaturas humanas: historia(6), literatura(4) y religión(2).
- Asignaturas de artísticas: música(8), dibujo(6) y expresión corporal(4).

MATRICES (Arrays bidimensionales)

85.- Realizar un programa que pida al usuario un numero N entre 3 y 5 y, a continuación, cree una matriz NxN (N filas, N columnas) cuyos elementos se le van a ir pidiendo al usuario.

86.- Crea un programa que cree una matriz de tamaño 5x5 que almacene los números del 1 al 25 y luego muestre la matriz por pantalla (en forma de matriz).

87.- Crea un programa que cree una matriz de 10x10 e introduzca los resultados de las tablas de multiplicar del 1 al 10 (cada tabla en una fila). Luego muestra la matriz por pantalla (en forma de matriz).

88.- Crea un programa que cree una matriz de tamaño NxM (tamaños introducido por teclado) e introduzca en ella los valores también a través del teclado teclado. A continuación deberá mostrar por pantalla cuántos valores son mayores que cero, cuántos son menores que cero y cuántos son igual a cero (es decir, se muestran 3 números).

89.- Crea un programa que encuentre el elemento de mayor valor de una matriz y…

a) … muestre la posición de su primera aparición.
b) … muestre una lista con las posiciones de todas sus apariciones.

90.- Dada una matriz MxM, crear un programa que muestre:

a) El mayor elemento de la diagonal principal.
b) La media de los valores de la diagonal principal.

91.- Se dice que una matriz M es simétrica perfecta si tiene el mismo número de filas y de columnas (MxM) y si todos sus elementos cumplen que el elemento de la posición [i] [j] es el mismo que el de la posición [j][i].
Implementa un programa que analice una matriz (creala y ponle valores tú) e indique si es simétrica perfecta o no.

92.- Implementar el juego de Piedra-Papel-Tijeras para dos jugadores humanos usando una matriz de comprobaciones: Cada fila corresponde a uno de los tres elementos de juego. Igual para las columnas. Las celdas de esa matriz tendrá algunos de los siguientes valores:

- 0 : si el elemento de la fila y el de la columna son el mismo (empate).
- 1: si el elemento de la fila gana al elemento de la columna.
- -1 : si el elemento de la fila pierde contra el elemento de la columna.

93.- La empresa MEDAC te ha elegido para crear un programa que se encargue de registrar los sueldos de los hombres y las mujeres que trabajan allí, con el fin de detectar si existe brecha salarial entre ambos.
Para ello, el programa pedirá primero el número total de trabajadores de la empresa por teclado (N) y, posteriormente, solicitará información de cada persona de la siguiente forma:

- pedirá su género (0 para varón y 1 para mujer)
- pedirá su sueldo anual en euros.

Esta información debe guardarse en la matriz. Finalmente, el programa mostrará por pantalla el sueldo medio de cada género.

94.- Vamos a crear la clase TABLERO (un tablero es una matriz MxM):

- `constructor 1:` necesitará dos valores: el número de filas y el número de columnas. Al crear un tablero, se rellenará de ceros todas sus celdas.
- `constructor 2:` necesita un solo valor. Ese valor será el que se usará tanto para las filas como para las columnas. Al crear un tablero, se rellenará de ceros todas sus celdas.
- `inicializarTablero:` rellena de ceros todas las casillas del tablero.
- `toString:` muestra el tablero por pantalla por filas y columnas. Sólo debe mostrar el contenido de las celdas del tablero (nada de comas, ni corchetes, ni caracteres raros). Además, si hay un 0 en una celda, se mostrará el carácter punto (.) y si hay un 1, el carácter asterisco (*)
- `rellenarPos(fila, columna):` coloca un 1 en dicha posición indicada en los argumentos.

95.- Crea un la clase MATRIZ con funciones para operar matrices NxN (simétricas). Los métodos que componen la clase son los siguientes:

- `constructor(int N):` crea una matriz de tamaño NxN y la rellena de ceros.
- `dimension`: devuelve el numero de filas y el numero de columnas de la matriz.
- `suma(matrizB):` devuelve una matriz con la suma de la matriz objeto con la matriz B que se pasa como argumentos. Para sumar dos matrices se suman los elementos de cada posición.
- `resta(matrizB):` devuelve una matriz con la resta de la matriz objeto con la matriz B que se pasa como argumento. Para restar dos matrices se restan los elementos de cada posición.
- `producto(num):` devuelve la matriz objeto con todos sus elementos multiplicados por el número indicado.

- `traspuesta(matriz)`: devuelve la matriz traspuesta de la matriz que se le pasa como argumento. La traspuesta de una matriz es la misma matriz pero cambiando filas por columnas (o viceversa).
- `producto(matrizB)`: devuelve una matriz obtenida de multiplicar la matriz objeto por la matriz B.

96.- El objetivo de este ejercicio es simular un cielo estrellado usando POO.

Para ello es necesario crear la **clase estrella**, la cual sólo necesitará unas coordenadas de posición dentro del cielo (X,Y). Aleatoriamente, también debe decidir su forma: Usaremos asterisco * para una estrella cercana y punto . para una estrella lejana. Intenta que la proporción de estrellas lejanas sea mayor que las cercanas (pej una proporción de 1 a 3).

El único método que tendrá esta clase es `toString()` el cual mostrará sólo la forma de la estrella.

La **clase cielo** la implementaremos usando una matriz. Para ello la clase recibirá el numero de filas y columnas. A continuación, creará un "cielo" vacío (la matriz sólo tendrá espacios en blanco).
Esta clase también dispondrá de un método llamado `ponerEstrellas` (público) el cual recibirá el numero de estrellas que se quieren colocar en el cielo y, posteriormente, colocará estrellas con posiciones aleatorias dentro del cielo (hay que comprobar que una estrella no se coloca en una posición ya ocupada).

También será necesario el método `mostrar`, el cual mostrará el cielo (la matriz) de manera correcta (por filas y columnas).
Para probar el funcionamiento, crea un cielo de 18 filas y 60 columnas, ponle 150 estrellas y muestralo.

97.- El cine Pingu te ha pedido que programes el software encargado de generar las entradas a las películas que se proyecten en su única sala.

Apartado A

Para ello es necesario crear la clase **Cine**. El cine va a disponer de una sala que se usará para cada sesión. La sala posee 50 asientos dispuestos en 5 filas con 10 asientos por fila. Cada asiento podrá tener un valor que indique que está libre u ocupado (tú decides ese valor. Me da lo mismo si usas números, caracteres, cadenas o booleanos). Obviamente, al comienzo de cada sesión todos los asientos deben estar vacíos.
El cine también tendrá como propiedades el precio de la entrada, el número de entradas vendidas y la recaudación total.

Los métodos que debe tener esta clase son:

- `constructor(double precio)`: recibe el precio NORMAL de la entrada, pone la recaudación y el numero de entradas a cero y crea una sala vacía.
- `calcularEntrada(String dia, Char carnet)`: devuelve el precio FINAL de una entrada tras aplicarle los siguientes descuentos:
  - Si el día es Martes o Jueves, el precio se reduce un 15%.
  - Si tiene alguno de los siguientes carnets:
    - Carnet de estudiante (E): aplica un descuento del 25% al precio de la entrada.
    - Carnet de jubilado (J): aplica un descuento del 50% al precio de la entrada.
    - Carnet de paro (P): aplica un descuento del 75% al precio de la entrada.
    - Si no se tiene ningún carnet (N), no hay descuento.
  - Es decir, primero se aplica el descuento del día de la semana (si corresponde) y sobre el nuevo precio de la entrada se puede aplicar el descuento de carnet (si tiene).
- `vaciarSala()`: pone todos los asientos de la sala en estado vacío.
- `mostrarSala()`: devuelve una cadena mostrando los asientos libres y ocupados en forma de matriz.
- `mostrarDatos()`: devuelve una cadena dónde se indica el precio NORMAL, el número de entradas vendidas y la recaudación que se lleva hasta ese momento.
- `cambiarPrecioNormal(double nuevo)`: cambia el precio normal de la entrada del cine.
- `buscarHueco(int personas)`: busca tantos asientos libres consecutivos en una misma fila como se indica en el argumento. Devuelve dos valores: la fila y la columna dónde empiezan el numero de asientos libres a ocupar por esas personas o los valores -1,-1 si no encuentra huecos libres.

Puedes crear otros métodos si lo crees que es necesario.

Apartado B

Tras reunirte con el encargado, habéis acordado que, haciendo uso de la clase realizada en el apartado A, se va a crear un archivo main que dispondrá de un menú general con tres posibles opciones:

A) *Vender Entradas:* Se pregunta por el numero de entradas que se desean comprar y se realizan los siguientes pasos:
   - Se debe comprobar que hay tantos asientos libres consecutivos en una misma fila como entradas se desean. Si no es así, se indica por pantalla y finaliza la venta (vuelve al menú y no se vende ninguna entrada).
   - Una vez que están localizados los sitios, se procede a vender las entradas. Por cada entrada solicitada se debe calcular el precio total por entrada.

RECUERDA: El precio total se calcula en base al día de la semana en que se compra y si la persona tiene alguno de los carnets que aceptan descuento

- Finalmente, se aumentará el numero de entradas vendidas, se mostrará el precio final a pagar (la suma de los precios totales de cada persona), se maracrán los sitios libres como ocupados y se aumentará la recaudación.

B) *Mostrar asientos:* genera un dibujo que muestra la cantidad de asientos libres y ocupados que tiene la sala actualmente.

C) *Resumen de ventas:* esta opción muestra el precio normal de la entrada, cuantas entradas se han vendido y cual ha sido la recaudación hasta el momento.

D) *Limpiar sala:* prepara la sala para una nueva sesión: *"rebobina la película, se recoge la basura de la sesión anterior"* y se vuelven a colocar todos los asientos como vacíos.

E) *Salir:* con esta opción se sale del programa.

98.- Autómata Celular Bidimensional: *El Juego de la Vida*.
El juego de la vida es un "juego" sin jugadores. Realmente es una simulación de como evoluciona una serie de células en un tablero durante un número de pasos. Esas células siguen una serie de reglas que indican en cada paso cuando deben morir o nacer.

Para implementar esta simulación vamos a usar un tablero de NxM (siendo N y M números enteros que pueden ser iguales o distintos). Estos valores deben declararse al principio del programa como constantes.

Cada celda del tablero va a contener la representación de una célula viva (1) o muerta (0). El tablero comenzará con un estado inicial (algunas células vivas y otras tantas muertas) que indicaremos nosotros a través de la función correspondiente.

Las células del tablero van a cambiar o no en base a sus 8 celdas vecinas y a una serie de reglas:

- Regla de nacimiento: Una célula nace si tiene exactamente tres vecinos vivos.
- Regla de supervivencia: Una célula permanece viva si tiene dos o tres vecinos vivos.
- Regla de aislamiento: Una célula muere o permanece muerta si tiene menos de dos vecinos vivos.
- Regla de superpoblación: Una célula muere o permanece muerta si tiene cuatro o más vecinos vivos.

La única interacción que tendrá el programa con el usuario es para pedirle el número de pasos que va a dar el autómata. Dicho número debe ser entero y positivo (hay que controlar esta condición).

El objetivo es implementar la clase ACB para simular el funcionamiento del autómata basándonos en toda la explicación anterior y usando como mínimo los siguientes métodos:

- `inicializarAutomata` : llena el autómata de células muertas.
- `mostrarEstado`: imprime el contenido del autómata teniendo en cuenta que mostrará las células vivas con el carácter '*' y las muertas con '.'
- `primeraJugada`: indica a través de código qué celdas tienen inicialmente células vivas.

ArrayLists

99.- Pide por teclado un número. Dicho número indica cuantos elementos se van a pedir a continuación. Almacena esas números en un ArrayList. A continuación:

a) Muestra el contenido de la lista.
b) Intercambia la primera posición con la última.
c) Pide otra lista de números distinta como se indica al principio y añade todos sus elementos a la lista inicial. *Nota: al final sólo debe existir un arrayList con todos los valores juntos.*
d) Calcula la suma de todos los elementos de la lista.
e) Calcula la media aritmética de los elementos de la lista.
f) Muestra todos los valores de la lista que sean menores a la media antes calculada.

100.- Reescribe la clase Pila del ejercicio 68 usando como estructura principal un ArrayList. ¿Que ventajas se obtienen frente a la creación de esa clase usando arrays normales?

101.- Crea un programa que, haciendo uso de un ArrayList de números enteros, muestre por pantalla un menú con las siguientes opciones:

1. *Agregar:* se le pide al usuario que indique un número entero por teclado y mete ese número dentro de la estructura.
2. *Buscar:* se le pide al usuario un número entero e indica si ese número se encuentra en la estructura o no.
3. *Eliminar:* se le pide al usuario un número entero y, si el elemento está en la estructura, lo elimina (y lo indica por pantalla). Si el elemento no está en la estructura, saca un mensaje indicándolo.
4. *Modificar:* se le pide al usuario un número entero y, si el elemento está en la estructura, se solicita otro número para cambiarlo. Si el elemento no está en la estructura, saca un mensaje indicándolo.
5. *Insertar en posición:* se le pide al usuario un número entero y una posición (entero mayor a 0). Mete el número en la posición indicada de la estructura.
6. *Mostrar:* muestra el contenido de la estructura.
7. *Salir:* Sale del programa y muestra un mensaje de despedida.

102.- Crea la clase **persona**. De un apersona necesitamos el DNI, nombre, apellido (uno solo), sexo, edad y peso. Todos privados.
La clase tendrá un constructor normal (con todos los argumentos) y uno solo con el DNI (nombre y apellido quedaran como cadena vacía, edad y peso se inicializará a 0.
Además de los getters y setters correspondientes, la clase tendrá los métodos: `toString` y `equals`.

Haciendo uso de la clase persona, implementa la clase **Cola de Supermercado**. En una cola de supermercado, la primera persona que entra en la cola, es la primera que puede abandonarla.

Usa las variables de clase que creas necesarias (todas privadas) e implementa los métodos:

a) `EstaVacia`: devuelve Verdadero o Falso si la cola está vacía o no.
b) `Entrar`: mete una persona en la cola.
c) `Salir`: saca de la cola a la persona que le corresponde. Muestra la información de esa persona.
d) `Primero`: este método devuelve una persona. La persona devuelta será la que está primera en la cola. Nota: No muestra nada por pantalla.
e) `Cantidad`: devuelve el número de personas que hay en la cola.

103.- Haciendo uso de la clase persona del ejercicio anterior, crea la clase **Agenda**, la cual tendrá una lista de personas (arrayList) y los métodos:

- `AniadirPersona`: recibe como argumento una persona y la añade a la lista.
- `BorrarPersona`: recibe como argumento un DNI y ,en el caso de que exista una persona con DNI, la borra. *Nota: borra la primera que encuentre.*
- `BuscarPersona`: recibe como argumento un DNI y muestra los datos de la persona con ese DNI. O un mensaje si ese DNI no está en la lista.
- `MostrarAgenda` (`toString`): muestra todos los datos de cada persona de manera ordenada.
- `OrdenarAgenda`: ordena las personas de la lista por apellidos. *Nota: para este punto debes buscar información sobre el método Collections.sort*

Crea un fichero **main** para probar la clase anterior. Para ello, muestra un menú para poder realizar las acciones antes indicadas y prueba su funcionamiento.

Arrays Asociativos/Diccionarios/Mapas

104.- Crea un programa (no clase) que vaya solicitando nombres de diferentes alumnos junto con su nota media del curso hasta que el nombre sea igual a "fin" (minúsculas). Almacena todos esos datos en una estructura tipo diccionario y muestra cada alumno con su nota en una línea.

A continuación sube un punto la nota de aquellos alumnos cuyo nombre empiecen por 'J' y borra aquellos cuyo nombre empiecen 'A' o por 'D'
Para finalizar, si la estructura no está vacía, muestra solo las notas y la media de todas ellas.

105.- Crea la clase ARAS la cual facilitará el trabajo con diccionarios de tipo String-String. Para ello es necesario implementar los siguientes métodos:

- `borrar(String clave)`: elimina la celda indicada por la clave.
- `meter(String clave,String valor)`: introduce una nueva celda en el diccionario con la clave y el valor indicados. Si ya existe una celda con esa clave, lo indica por pantalla y no toca el diccionario.
- `actualizar(String clave,String valor)`: busca la celda indicada por la clave y cambia su valor. Si la celda no existe, lo indica por pantalla.
- `existeClave(String clave)`: devuelve verdadero o falso dependiendo de si existe o no la celda que indica la clave que se pasa en el argumento.
- `existeValor(String valor)`: devuelve verdadero o falso dependiendo de si existe o no alguna celda con el valor que se pasa en el argumento.
- `primerValor(String valor)`: devuelve la clave de la primera celda que tenga el valor pasado como parámetro. Si no existe ninguna clave, devolverá la cadena `"undefined"`.
- `mostrar(diccionario)`: muestra el diccionario con sus pares en columna y ordenados por clave.

```
Clave: ……  Valor: ……

Clave: ……  Valor: ……

Clave: ……  Valor: ……

Clave: ……  Valor: ……
```

106.- Crea una clase para almacenar las ciudades de nacimiento y los nombres de cada alumno/a de un instituto. *IMPORTANTE: la clase almacenará solo tipos básicos (String en ambos casos). No hay que crear la clase Alumno (ni ninguna otra salvo la principal) en este ejercicio*.

Crea las variables de clase que necesites (todas privadas) e implementa los siguientes métodos:

- `aniadirAlumno(String nombre, String ciudad)`: almacena el par nombre-ciudad en la estructura. Si existe ya un alumno con ese nombre, se indica por pantalla y no se almacena nada.
- `mostrarCiudad(String ciudad)`: Si se encuentra en la estructura, muestra la ciudad que se le pasa como parámetro y el número de alumnos que han nacido en ella.
- `cantidadCiudades()`: devuelve el número de ciudades distintas que hay en la estructura.
- `mostrarDatos()`: devuelve una cadena compuesta de cada alumno y su ciudad de nacimiento.
- `borrar(String nombre)`: si existe, elimina al alumno con ese nombre. Si no es así, indica un mensaje por pantalla.
- `borrar()`: elimina todos los alumnos.
- `resumen()`: devuelve una cadena donde se indica cada ciudad almacenada y el número de alumnos/as que han nacido en ella.

Ejemplo:

```
Málaga: 23 alumnos.
Granada: 12 alumnos.
Jaén: 5 alumnos.
Madrid: 10 alumnos.
...
```

Crea un fichero **main** para probar la clase anterior. Para ello, muestra un menú para poder realizar las acciones:

1. Añadir Alumno.
2. Borrar Alumno.
3. Mostrar Instituto.
4. Alumnos por ciudad.
5. Mostrar datos. *(Esta opción debe mostrar el numero de ciudades distintas que hay y el numero de alumnos por ciudad)*
6. Borrar todo.
7. Salir

107.- Implementa un programa que cuente el numero de apariciones de cada palabra de un texto. Para ello se debe crear una estructura que almacene cada palabra y el número de veces que aparece en el texto.
Recuerda que un texto puede tener entre 0 y N palabras (siendo N un número desconocido) y que una palabra puede estar escrita en mayúscula, minúscula o combinación de ambas (y sigue siendo la misma palabra)

Para este ejercicio, el texto va a ser introducido línea a línea hasta por el usuario hasta que se introduzca una línea vacía.
Tras eso deben mostrarse todas las palabras del texto una debajo de otra junto con el número de veces que aparece cada una.

A tener en cuenta:

- Hay que usar funciones para trabajar con cadenas (¿Te acuerdas del primer trimestre?) Te recomiendo que uses `split()`. Te va a ser muy útil en este ejercicio.
- Vamos a suponer que no se introducen signos de puntuación en el texto. Sólo letras,números y espacios en blanco.

108.- Realiza cada una de las tres partes que se indican a continuación:

PARTE 1

Crea la clase **Empleado**. De un empleado necesitamos su nombre, su apellido (uno solo), sexo, el sueldo base que tiene y los años de antigüedad en al empresa. Todos privados.
La clase tendrá un constructor con el nombre, el apellido y el sexo. El sueldo base será de 800€ y los años de antigüedad será un número aleatrio entre 0 y 10.

Además de los getters y setters que necesites para resolver la segunda parte de este ejercicio, la clase tendrá los métodos: `toString` y `equals` (tú decides cuando un empleado es igual a otro).

PARTE 2:

Haciendo uso de la clase empleado del párrafo anterior, crea la clase **Departamento**, la cual se compondrá de varios empleados almacenados en una estructura según su DNI (tipo Integer). *Nota: la clase empleado no almacena el DNI, por tanto NO PUEDES definir el DNI en dicha clase.*

También son necesarios los siguientes métodos:

- `AniadirEmpleado`: recibe como argumento un DNI y un empleado y lo añade a la estructura.
- `BorrarEmpleado`: recibe como argumento un DNI y ,en el caso de que exista un empleado con ese DNI, la borra.
- `BuscarEmpleado`: recibe como argumento un DNI y muestra los datos del empleado con ese DNI. O un mensaje si no existe un empleado con ese DNI.
- `RevisarSueldos`: este método recorre toda la estructura y actualiza los sueldos según el número de años de la empresa:
  - Entre 0 y 1, no toca el sueldo.
  - Entre 2 y 4, aumenta 200€.
  - Entre 6 y 8, aumenta 500€.
  - Entre 9 y 10, aumenta 750€.
  - Más de 10, aumenta 900€.
- `NuevoAnio`: recorre la estructura y suma un año a todos los empleados.
- `MostrarDepartamento`: muestra todos los datos de cada empleado de manera ordenada.

PARTE 3:

Crea un fichero **main** para probar la clase anterior. Para ello, muestra el siguiente menú:

1. Mostrar Departamento.
2. Añadir Empleado.
3. Despedir Empleado.
4. Aumentar antigüedad al departamento.
5. Revisar salarios.
6. Despedir a todos los empleados.
7. Salir

Programa la funcionalidad de cada entrada del menú usando la clase Empleado.

Excepciones

Hay que usar manejadores de excepciones para controlar los posibles problemas que puedan surgir a la hora de hacer cada ejercicio a partir de este momento

109.- Crea un programa que pida al usuario una cadena de texto y un numero entero N. A continuación, el programa indicará el carácter de la cadena situado en la posición N. En caso de que no se pueda realizar dicha acción debe mostrarse el mensaje: *"No existe la posición N en la cadena" (Siendo N el número indicado por el usuario)*.

110.- Crea la clase Impar. Dicha clase tendrá como variable de clase `numero` (un número entero). Además, los métodos que va a tener la clase son:

- `constructor:` Recibe como argumento un número entero. Si el argumento es un número par, lanza una excepción con un mensaje indicando que el número no es impar. En caso contrario, asigna ese número a la variable de clase.
- `toString:` muestra el numero que tiene almacenada la variable de clase.

Crea un archivo MAIN que sea capaz de usar correctamente la siguiente línea de código: `System.out.println(new Impar(24))`

111.- Crea un programa que lance y capture una excepción del tipo `RuntimeException`. Para ello el programa hará lo siguiente:

a) Mostrar el texto: *Programa automático*
b) Crear una excepción del tipo indicado con el mensaje: s*oy una excepcion*.
c) Lanzar la excepción (throw).

Añade el código necesario para que, pase lo que pase, se muestre el mensaje: *Programa terminado.*

112.- Desarrolla el juego 'Adivina el número' (ejercicio 45 de esta relación de problemas) pero controlando que se introducen números.

Recordatorio de las reglas:
El ordenador debe generar un número entre 1 y 100, y el usuario tiene que intentar adivinarlo. Para ello, cada vez que el usuario introduce un valor, el ordenador debe decirle al usuario si el número introducido es mayor o menor que el pensado. Cuando consiga adivinarlo, debe indicárselo e imprimir en pantalla el número de veces que el usuario ha intentado adivinar el número.

Si el usuario introduce algo que no es un número, debe indicarlo en pantalla, y contarlo como un intento.

113.- Sea la clase piscina definida como sigue:

```
public class Piscina{

	public final int MAXNIVEL;
	private int nivel;

	public Piscina(int tope){
		if (tope<0){
			tope=0;
		}
		this.MAXNIVEL = tope;
	}

	public int getNivel(){
		return this.nivel;
	}

	public void vaciar(int cantidad){
		this.nivel -= cantidad;
	}

	public void llenar(int cantidad){
		this.nivel += cantidad;
	}
}
```

a) Modifica los métodos vaciar y llenar para que lancen una excepción:

- vaciar: si la piscina queda por debajo de 0.
- llenar: si la piscina queda por encima del nivel máximo.

b) Programa un archivo main que se encargue de crear una piscina con un valor aleatorio entre 1 y 100. A continuación, en un bucle FOR de 5 vueltas, llena la piscina con un numero aleatorio entre 1 y 25 mostrando el nivel de la piscina en cada vuelta. Seguidamente, haz lo mismo pero vaciando la piscina (mismas vueltas y números aleatorios)

114.- Escribe un programa que lea dos números por teclado, numerador y denominador respectivamente. A continuación se debe comprobar que numerador es menor que 100 y denominador mayo de -5. Si esto no es así se lanzará una excepción de tipo Exception. A continuación se calculará el cociente y se mostrará por pantalla.

Ten en cuenta que, aparte de la excepción indicada, también se puede producir una división entre cero y también que al pedir los números por teclado se introduzcan caracteres no numéricos.

115.- ¿Qué se obtiene al ejecutar estos códigos?

```
//APARTADO A
public class PruebaExcepciones{
      public static void maín (String [] args) {
            try{
                  int s=4/0;
                  System.out.println ("El programa sigue");
            }

            catch (ArithmeticException e){
                  System.out.println ("División por 0");
            }

            catch (Exception e){
                  System.out.println("Excepción general") ;
            }

            System.out.println ("Final del main");
      }
}
```

```
//APARTADO B
public class PruebaExcepciones{
      public static void maín (String [] args) {
            try{
                  int s=4/0;
                  System.out.println ("El programa sigue");
            }

            catch (ArithmeticException e){
                  System.out.println ("División por 0");
            }

            finally{
                 System.out.println ("Soy el finally") ;
            }

            System.out.println("Final del main");
      }
}
```

116.- Suponiendo que existe el siguiente método:

```
public int accesoIndice(int [] lista, int j)
   try{
     if ((j >= 0) && (j<=lista.length)){
        return lista[j];
      }
      else{
        throw new RuntimeException("El indice "+j+ "
         no existe en el array");
      }
   }
}
```

¿Que se obtiene al ejecutar el siguiente código?

```
Public static void main(String [] args){
   int [] lista = new int[15];
   accesoIndice(lista,15);
}
```

¿Sería necesario mejorar ese código? ¿Por qué?

Lectura y Escritura de ficheros

117.- Crea un programa que vaya pidiendo frases por teclado al usuarios hasta que introduzca una frase vacía.
Para cada frase que el usuario introduzca, se almacenará una debajo de otra en un fichero de texto llamado `EJ117.dat` Este fichero no debe existir al ejecutarse el programa.

118.- *(Otra versión del ejercicio 107 de la relación de dicccionarios)* Dado el fichero `EJ118.txt`, el cual contiene un texto de varias líneas, implementa un programa que cuente el numero de apariciones de cada palabra de ese fichero.
Para ello crea una estructura que almacene cada palabra y el número de veces que aparece en el texto.

Recuerda que un texto puede tener entre 0 y N palabras (siendo N un número desconocido) y que una palabra puede estar escrita en mayúscula, minúscula o combinación de ambas (y sigue siendo la misma palabra)

La parte final del programa mostrará todas las palabras del texto una debajo de otra junto con el número de veces que aparece cada una.

A tener en cuenta:

- No importa el orden en el que se almacenen/muestren las palabras del texto.
- Vamos a suponer que no hay signos de puntuación en el texto. Sólo letras, números y espacios en blanco.

119.- Implementa un programa que, dado un fichero de texto, muestre todas sus lineas precedidas por el numero de linea correspondiente y el carácter dos puntos (:)

120.- Implemente un programa que muestre la línea más larga de un fichero y:

a) Si hay más de una, sólo muestre la primera.
b) Si hay más de una, sólo muestre la última.
c) Si hay más de una, muestre todas.

121.- Realiza un programa que dado un fichero de texto, muestre el numero de caracteres, el numero de palabras y el número de líneas del fichero.

122.- Supongamos un fichero de texto que se usa como inventario, el cual tiene los datos en cada línea estructurados de la siguiente forma: `nombre:precio:cantidad`

Ejemplo:

```
amuleto:200:5
pocion:50:20
espada:1000:1
antorcha:30:9
cuerda:120:3
```

Diseña un programa que, tomando un fichero con la estructura antes indicada, muestre por pantalla de forma clara el inventario que representa:

```
Item:     amuleto
Precio:   200
Cantidad: 5

Item:     pocion
Precio:   50
Cantidad: 20
...
```

123.- Realiza un programa en el que se le solicite al usuario el nombre de dos archivos de texto que existan (Pej: `datos.txt y nombres.txt`) A continuación, el programa creará un nuevo archivo cuyo nombre será el nombre de los archivos anteriores separados por un guion medio y extensión .txt (Pej: *según el nombre de los archivos en el anterior ejemplo, el nuevo fichero se llamará:* `datos-nombre.txt`)
En ese fichero nuevo se copiará todo el contenido del primer fichero indicado y, a continuación, el contenido del segundo fichero indicado.

124.- Modifica el ejercicio 106 de la relación de Diccionarios para que, al ejecutar el programa lea y cargue en memoria (en una estructura) los datos de un fichero de texto llamado `datos.dat` con el siguiente formato: `alumno: ciudad`

Pej;

```
Jaime: Jaen
Cristina: Cadiz
Irene: Malaga
Pablo: Jaen
Juanje: Malaga
```

A continuación mostrará el menú con dos nuevas opciones:

1. Guardar instituto: pasa toda la información almacena en la estructura al fichero `datos.dat`
2. Cargar instituto: carga los datos del fichero `datos.dat` y machaca con esos valores la estructura.

Además, al salir del programa SIEMPRE se va a guardar el contenido de la estructura en el fichero `datos.dat`

Importante: El resto de opciones siguen funcionando normalmente. Se guardarán/borrarán alumnos en la estructura, no en el fichero. El fichero sólo se actualizará cuando se elija la opción de Guardar o se salga del programa.

125.- Se quiere implementar un juego de apuestas para un jugador llamado *Horse Alone* cuyo **funcionamiento** es el siguiente:

a) Al empezar, aparece un mensaje de bienvenida indicando el nombre del juego.
b) Se le pregunta el nombre al jugador.
c) El programa buscará el archivo 'nombre del jugador' con extensión `.dat` en el directorio *data*.
    1. Si no existe, lo crea y lo rellena sus campos con valores por defecto. *(ver el formato más adelante)*
    2. Si existe, carga los datos de los campos de ese fichero. *(ver el formato más adelante)*
d) Tras cargar o inicializar los datos, se muestran esos datos por pantalla de forma clara y ordenada.
e) El programa debe cargar también el archivo `horses.dat` *(ver el formato más adelante)* y leer su contenido. Si el archivo no existe dará un error, indicará que las carreras no pueden realizarse porque no hay caballos y finalizará todo.

f) Zona de la carrera: Al empezar a jugar, se deben mostrar los datos de los tres caballos que compiten (nombre y victorias) y se le pedirá al jugador que apueste por uno de los tres. Hay que controlar que se introduce un valor relacionado con uno de los tres caballos.
g) A continuación se le pide una apuesta al jugador. Dicha apuesta debe ser un valor mayor a 15€. Hay que evitar meter valores erróneos (números inadecuados, cadenas…). Si el jugador tiene menos de 15€, no podrá apostar. Se le indicará y el programá se cerrará.
h) Si el valor de la apuesta es correcto, se le pregunta al jugador si está seguro. Si la respuesta es negativa, se vuelve a pedir la apuesta. Si es positiva, se pasa a la a la carrera.

i) El programa decidirá de forma aleatoria qué caballo es el ganador y mostrará por pantalla el resultado (lo ideal es que se pongan algunas frases relatando la interesantísima carrera y su ganador).
j) Si el gana el caballo apostado por el jugador, este consigue una ganancia del 150% de su apuesta inicial. Sino, pierde lo apostado.

k) En este punto el programa debe actualizar los datos que maneja:
   1. Actualiza el número de victorias del caballo ganador.
   2. Del jugador, actualiza el número de partidas jugadas, el dinero y el número de partidas ganadas si procede.

l) Si el jugador tiene más de 15€, se le pregunta si quiere seguir apostando. Si responde afirmativamente, se vuelve a la zona de la carrera. Sino, se cierra el programa.
m) Si el jugador tiene menos de 15€ se le indica que ha perdido y se cierra el programa.

Siempre que se cierre el programa, se deben guardan todos los datos en los ficheros correspondientes

Formato de los ficheros de datos

Jugador

Debe tener los valores referentes al: nombre, dinero, apuestas totales y apuestas ganadas. Cada campo irá con su valor correspondiente en una línea. Ejemplo:

`jaime.dat`

```
nombre:Jaime
dinero:735
aput:56
apug:47
```

Caballos

Cada línea del fichero tendrá el nombre del caballo y su número de victorias. Ejemplo:

`horses.dat`

```
Drogadicto:17
Galleta:26
Illo:13
```

NOTA: Se pueden usar otros formatos de texto parecidos (deben aparecer los campos indicados) que se ajusten mejor a las estructuras elegidas para ser usadas en el programa.